PMaC Binary Instrumentation Library for PowerPC/AIX

Mustafa M. Tikir, Michael Laurenzano, Laura Carrington, Allan Snavely

Performance Modeling and Characterization Lab
San Diego Supercomputer Center
9500 Gilman Drive, La Jolla, CA 92093

{mtikir,michaell,lcarring,allans}@sdsc.edu

Abstract

The decommissioning of Alpha AXP-based systems carrying the ATOM toolkit has left the need for an efficient, flexible binary instrumentation tool-building framework for another platform. PMaCinst is a binary instrumentation toolkit that operates on XCOFF binaries on AIX for PowerPC processors. PMaCinst has a C++ API that provides the means to inject code and data into a binary file In this paper, we first present the mechanisms for performing these modifications along with the key parts of the API. We then present three example rewriting tools that have been built using PMaCinst, which help to highlight some of the correctness and efficiency issues that can be encountered by tool writers. Finally, we show that programs instrumented with PMaCinst slow down at rates that are comparable to equivalently instrumented programs created with ATOM.

1. Introduction

Program instrumentation tools [1-3] provide a unique and important source of information for understanding and tuning the execution behavior of applications. These tools have proven to be effective during every stage of an application's development cycle. They have been extensively used to evaluate how a program will perform on new systems, to identify performance bottlenecks during program execution, to gather information on underlying systems for profile-driven optimizations, and to identify bugs in programs [9].

In the Performance Modeling and Characterization (*PMaC*) Lab at the San Diego Supercomputer Center, we focus on developing methods and tools for understanding and predicting the performance of scientific applications on existing and future HPC systems. One of these tools, called the MetaSim Tracer[4], provides the user with a detailed summary of the fundamental operations carried out by the application. MetaSim Tracer is a binary rewriting tool currently implemented on top of the ATOM toolkit [1] for Tru64 Unix on Alpha AXP processors. However, due to the decommissioning of current Alpha-based systems, the need has emerged for an efficient, flexible instrumentation toolkit that operates on a different platform.

In this paper we introduce *PMaCinst*, the PMaC binary instrumentation library. PMaCinst enables instrumentation of XCOFF [5] executables on the AIX operating system on PowerPC processors [6]. PMaCinst is a tool-building system similar in nature to popular toolkits such as Dyninst [2], ATOM and Pin [3]. It is designed to provide functionality that is similar to these tools in that it allows arbitrary functions to be inserted at arbitrary points in executables. Furthermore, it provides the facilities to insert hand optimized low-level assembly code, which can help keep instrumentation overhead minimal. PMaCinst works on any compiled XCOFF binary (both 32 and 64-bit), independent of the compiler and language system used to generate the binary.

We also present several rewriting tools implemented using the PMaCinst library. These tools include a basic block tracing tool that prints function name and line number information of basic blocks as they are executed, a basic block counting tool that counts the number of times each basic block is executed during a program run, and a cache simulation tool that can simulate multiple memory hierarchies with different numbers of cache levels using the address stream of the program. The PMaCinst instrumentation library source code and the previously mentioned rewriting tools are available for download at http://www.sdsc.edu/PMaC/PMaCinst/.

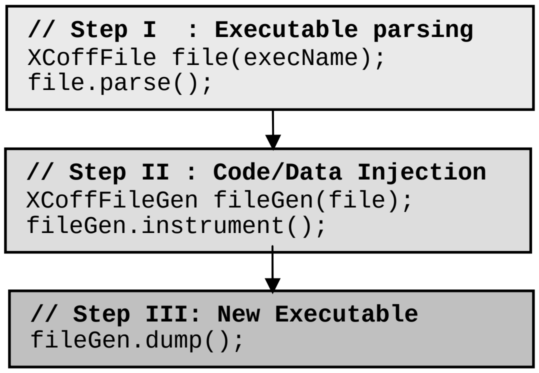
2. PMaCinst API

PMaCinst library provides an Application Programmer Interface (API) in C++ for building program analysis and rewriting tools. This API includes functionality to parse an XCOFF executable and generate program objects, to inject instrumentation code and data into the executable, and to generate a modified XCOFF executable that includes instrumentation code and the accompanying changes to program objects. The general framework for PMaCinst is shown in Figure 1.

2.1 Parsing the Executable

The PMaCinst API provides class definitions for XCOFF file objects such as the file header, section header, symbol table, relocation table and line information table, and provides class definitions for program objects such as instructions, basic blocks, functions, control flow graphs (CFG), memory operations, and natural loops.

The `parse` method of the `XCoffFile` class (shown in Figure 1) parses an input executable file, creates XCOFF objects, and creates program objects for the entire executable. The parse method parses every object at startup instead of incremental parsing, where program objects would be parsed as needed with a potentially smaller overhead. Parsing every object at startup allows PMaCinst to assign a unique and persistent ID, called a *hashcode*, to each program object encountered in the executable. This hashcode is based on a hierarchy of container objects (such as instructions in a basic block, basic blocks in a function, functions in a code section, etc.), and parsing the executable in its entirety at startup enables PMaCinst to assign the same hashcode to each object every time the executable is parsed. Unlike parsing, much of the remaining functionality of PMaCinst is invoked on demand, such as constructing loops or generating line number information for a particular function or for all functions in a code section.



```
// Step I  : Executable parsing
XCoffFile file(execName);
file.parse();
```

```
// Step II : Code/Data Injection
XCoffFileGen fileGen(file);
fileGen.instrument();
```

```
// Step III: New Executable
fileGen.dump();
```

Figure 1. General PMaCinst Framework

Executable parsing is fairly straightforward as XCOFF is a well defined and self contained object file format. However, special processing is required in two notable cases: identifying the sizes of some functions, and generating the CFG for functions with indirect jumps resulting from high-level constructs such as switch statements (multi-target jumps that use lookup tables).

PMaCinst does not require that executables contain debugging information. The minimal symbol table for an XCOFF file, which is generated when the program is compiled without the debug flag (e.g. –g), provides information about the sizes of all *control sections* (CSECT [5]) in the executable. It also provides size information for some functions. However, a CSECT may contain several functions for which the symbol table only provides information about the containing CSECT. For these functions, we use the start address of the function's *traceback table* to determine where the function ends. On AIX, a traceback table for each function is generated by the compiler for exception handling and inter-language call mechanisms, and is placed the end of the function's address space. Each traceback table starts with a null word (with a value of 0 also indicating an invalid instruction), which can then be used to mark the end of its accompanying function.

To generate the CFG for a function with indirect jumps, we first identify whether the jump instruction is a multi-target lookup table jump generated by a high level code construct such as a switch statement or whether it is simply an indirect call to a function. To identify multi-target jumps, we search for compiler-specific instruction sequence patterns generated for lookup table jumps similar to general peephole optimization techniques used in compilers. Such patterns contain instruction sequences to compare the switch value with the lookup table size as well as instruction sequences to calculate the jump addresses. Using the information available in these instruction sequences, we identify the locations of lookup tables in the executable as well as the number of entries in them. Later we parse the lookup table to generate the potential target addresses for the indirect jump instruction.

Since we search for instruction sequence patterns to identify the multi-target jump instructions, PMaCinst requires prior knowledge about all potential patterns that may be generated by the available compilers on the system. In this early version of the PMaCinst library, we handle common patterns for the GNU and native compilers for C, C++, and Fortran. We plan to extend these patterns in future versions of PMaCinst to include other languages and compilers.

PMaCinst also provides on-demand methods that discover and interface to more complex program information such as dominator information, natural loops and line number information. For loop discovery, we use the algorithms presented in [12] along with the linear-complexity dominator finding method presented by Lenguar and Tarjan in [13].

Necessary line information can be found in the line information and symbol tables of the XCOFF executable when debugging information is present, but finding relevant information from these sources can involve time-wasting searches. For example, determining the source file from which a particular instruction came from would typically involve searching the line information table to determine which symbol (function) is associated with that instruction then searching backwards from this function in the symbol table to find the file symbol that contains the function. Such searching is inefficient and can be incredibly redundant when many such lookups are being done. Instead of working from scratch for each such search, the user can invoke the `setLineInfoFinder` method of the `XCoffFile` class to create data structures that maintain intermediate information about how the virtual addresses of instructions, line information, function names and file names are related to one another. Returning to our example, storing a pointer to the parent file symbol for each function would save us the time of searching the symbol table for the containing symbol each time we performed an address to function name lookup.

2.2 Modifying Program Objects

To perform actual instrumentation after the executable is parsed, the PMaCinst API provides the user with a base class called `XCoffFileGen` (shown in Figure 1). This class provides virtual methods which must be implemented for every new rewriting tool. It also hides other abstractions that are common to all instrumentation frameworks. The virtual methods in `XCoffFileGen` include methods to select instrumentation points, print information about each point, calculate the size and generate the code for the point, query the name of the shared library and its functions for which the calls will be injected, and extend and initialize the DATA section of the program. After implementing these virtual methods, the `instrument` method of `XCoffFileGen` needs to be called by the new tool to apply all of the necessary modifications to the affected program objects. Several examples of rewriting tools and how this base class is extended are provided under the `examples/` directory of the PMaCinst source distribution.

PMaCinst provides two different ways of injecting instrumentation code at an arbitrary instrumentation point. First, a sequence of assembly instructions can be inserted. This mainly enables users to insert hand optimized code. Second, calls to functions from a shared library can be inserted. In order to insert calls to the functions from shared libraries, PMaCinst requires that intermediate *call stubs* be generated to handle the saving/restoring of registers before/after function calls to preserve machine state as well as the passing of parameters to the library functions through registers.

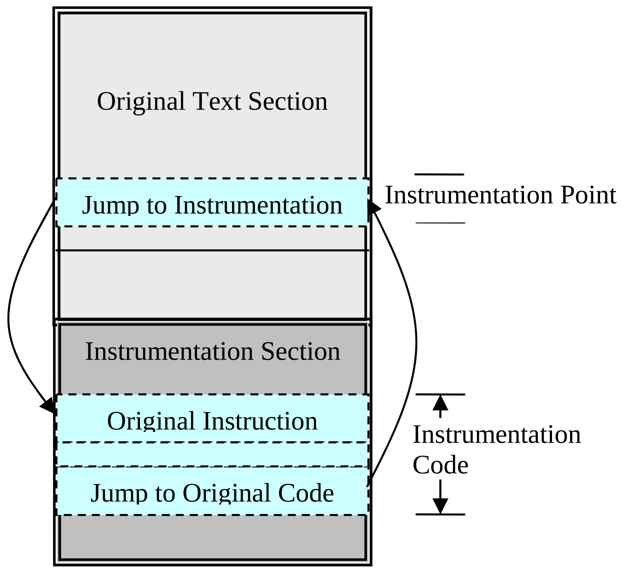


Figure 2. Instrumentation in PMaCinst

In both types of code insertion, instrumentation code for each instrumentation point in a text section is placed at the end of the text section, meaning that the address space for the original code is left unperturbed (except for the single replaced instruction at the instrumentation point). Alternatively, instrumentation code could be inlined at each instrumentation point, but such inlining would require updating the offsets of all branch and call instructions throughout the executable, making it extremely difficult to ensure correctness of the instrumented executable as well as potentially mitigating any code layout optimizations done by the compiler.

Control into the instrumentation code is acquired by relocating an instruction from the instrumentation point in the original code into the instrumentation code, then replacing the original instruction with a jump into the instrumentation code. This is similar in concept to a base trampoline in the Dyninst library [2] and can be seen in Figure 2. At the end of each instrumentation code, a jump back to the original code is inserted.

In PowerPC, a relative jump instruction can jump a distance of +/- 32MB from the instruction itself. Thus if instrumentation for a point requires jumping more than this maximum distance, PMaCinst denies the instrumentation. However, such cases are rare as typical TEXT sections in executables tend to be smaller than the maximum range of the relative jump instructions. Alternatively, for such cases, indirect jumps based on the control and link registers can be used to branch into the instrumentation code, which requires determining which registers are live at instrumentation points in order to keep the machine state unchanged. We plan to include this feature in future versions of PMaCinst.

Since PMaCinst enables insertion of assembly code, developers are currently expected to have some knowledge of the PowerPC instruction set and assembly code. Unlike Dyninst, we do not provide the ability to generate assembly code from high level C-like statements. Instead, for high level code, PMaCinst enables insertion of calls to shared library functions similar to ATOM. We chose to involve developer more because small sequences of hand written code are typically more efficient than automatically generated code. It allows us to insert manually optimized assembly code and to use efficient instrumentation code in which only the registers that are used in the instrumentation code are saved. However, in future versions of PMaCinst, we plan to include a higher level interface that will allow the user to generate and insert code that is given as C-like statements.

In addition to allowing insertion of instrumentation code, PMaCinst also provides interface to extend the initialized data section (DATA) and to initialize the extended space. This interface requires the user to define the size of the extension in bytes, and the structure of the additional space is unknown to PMaCinst. Rather, the user is required to know the structure of the additional space and how to use this structure during initialization and while accessing this space from the instrumentation code.

Similar to code insertion, the additional data is injected at the end of DATA section. Thus the original data in this section is left unperturbed. In XCOFF executables, the Table Of Contents (TOC [5]), which is accessed via the TOC register, is kept at the end of DATA section. Since PMaCinst stores additional data at the end of the DATA section, it follows that both the TOC and the extended part of the DATA section can be accessed via TOC register in the instrumentation code. This allows users to keep track only of offsets to their new data with respect to TOC register rather than the absolute addresses of the data.

2.3 Writing the Instrumented Executable

To generate the instrumented executable file, the `XCoffFileGen` class provides a method called `dump` (shown in Figure 1). This method iterates over all program objects and generates XCOFF file objects from them, then dumps these newly created XCOFF file objects to an executable file that contains the instrumented program. It is important to note that the newly created XCOFF executable is run in similar fashion to the original executable without any additional processes or intermediate files to pass information (unless, of course, any of the instrumentation code inserted by the user reads or writes on files).

Even though PMaCinst keeps the original data and TEXT sections unperturbed except relocating instructions at instrumentation points, special handling is required to generate a correct XCOFF executable if the data section of the original executable has been extended. The un-initialized data section (BSS [5]) generally follows the DATA section in the virtual address space of the program. Furthermore, the TOC in the DATA section includes pointers to items in the BSS and TEXT section. Thus, for program correctness the pointers to BSS objects need to be updated when the DATA section (and thus the virtual address of the BSS section) is extended. But since the TEXT section generally lies before the DATA section, the TOC pointers to the TEXT section remain unchanged.

3. Rewriting Tools and Experimental Results

We implemented three rewriting tools using the PMaCinst instrumentation library. The *BasicBlockTracer* prints the sequence number, the line number, and the function name of a basic block as that block is executed. The *BasicBlockCounter* counts the execution frequency of each basic block during a program run, printing the execution counts of all blocks at the end of execution. The *CacheSimulator* simulates multiple memory hierarchies using the address stream of some basic blocks where each hierarchy may include varying levels of caches.

3.1 Example 1: BasicBlockTracer

The BasicBlockTracer tool selects the entry points of all basic blocks in the executable for instrumentation. To be able to print information for each basic block, the BasicBlockTracer extends the DATA section and initializes it with a sequence of records where each record contains information about a basic block that will be passed to the instrumentation function. Each record contains the block sequence number (a unique ID assigned at instrumentation point selection), line number, and function name of the basic block.

At the entry point of each block, BasicBlockTracer inserts a call to a function in a shared library, which takes a pointer to that basic block's record in the DATA section as an argument and prints the information to standard output. Since BasicBlockTracer prints the sequence number of basic blocks rather than the virtual addresses, it also writes information about mapping between sequence numbers and virtual addresses of instrumented basic blocks to a file while instrumentation is being performed, which may be used for post-processing if desired.

Figure 3 shows a sample output of an executable instrumented by BasicBlockTracer. The base printed for each basic block is the virtual address of that block's record in the DATA section. Note that for BasicBlockTracer to print source line numbers for basic blocks, it is required that the program be compiled with debug information (e.g. –g).

```
Base 0x11000150c, Block 66 is at line 10 in function .foo
Base 0x11000153c, Block 69 is at line 17 in function .foo
Base 0x11000154c, Block 70 is at line 17 in function .foo
Base 0x11000155c, Block 71 is at line 17 in function .foo
Base 0x11000156c, Block 72 is at line 22 in function .foo
Base 0x1100013d4, Block 47 is at line 5 in function .main
Base 0x1100015a0, Block 75 is at line 1 in function .bar
Base 0x1100015b0, Block 76 is at line 4 in function .bar
Base 0x1100015c0, Block 77 is at line 7 in function .bar
Base 0x1100015d0, Block 78 is at line 8 in function .bar
Base 0x1100015e0, Block 79 is at line 9 in function .bar
Base 0x11000158c, Block 74 is at line 1 in function .bar_helper
```

Figure 3. A sample output from the BasicBlockTracer

3.2 Example 2: BasicBlockCounter

Like, BasicBlockTracer, the BasicBlockCounter selects the entry point of every basic block as an instrumentation point. To keep track of execution frequencies of basic blocks, the BasicBlockCounter extends the DATA section with an array of 64-bit counters that are indexed by the block's sequence number and initialized to 0.

For each basic block, the BasicBlockCounter tool inserts assembly code that increments the counter for that basic block. In addition, BasicBlockCounter inserts instrumentation at the entry point of the exit function of the program that writes the collected basic block execution frequencies into a file at program termination. The instrumentation code at the exit function includes a call to a function in a shared library, which is passed a pointer to the counter array and the size of the array as arguments. At program termination, the tool writes the sequence number and execution frequency of each basic block instrumented in the program.

To relate the sequence numbers that are printed to unique hashcodes for the instrumented basic blocks, BasicBlockCounter also writes information about the blocks to an output file during instrumentation, which includes the hashcode, sequence number, virtual address, function name, and source file name of the basic block. Again, users could use this extra information to associate all of the statically known information about a basic block listed above with the collected runtime execution frequencies.

3.3 Example 3: CacheSimulator

The CacheSimulator tool selects all of the memory instructions from a user-defined list of basic blocks as instrumentation points. Suppose that we select the list of the most frequently executed basic blocks that correspond to 95% of total program execution. This list of basic blocks can be obtained by instrumenting the application with the BasicBlockCounter tool.

A simple way for CacheSimulator to instrument a memory instruction is to insert a call to a shared library function and pass the effective address of the memory instruction as an argument so that the function could simulate this address being accessed in a series of caches. But such instrumentation is computationally expensive, as the number of memory operations in a program tends to be large and calling the simulation function for each memory operation separately will introduce a significant amount of function call overhead. Instead, CacheSimulator first extends the DATA section with a space that will act as a buffer for memory operation records, where each memory operation record is composed of the effective address, basic block sequence number, and the index of the memory instruction within the block.

The instrumentation code inserted by CacheSimulator for each memory instruction stores information about the effective address and the location of the instruction itself in the first available record in the address buffer and increments the buffer size. When the buffer is full, the instrumentation code calls the cache simulation function from a shared library, passing the start address of the buffer as an argument. Like BasicBlockCounter, CacheSimulator also inserts instrumentation code at the entry of the exit function of the program to write the results of the cache simulations, which is a call to another function in the shared library.

By storing the program's address stream in a buffer and calling the cache simulation function when the buffer is full, the simulation function is called less frequently (every N memory accesses, where N is the size of the buffer and can be given as a parameter to the CacheSimulator tool). This framework also enables the user to plug in new cache simulation functions without requiring re-instrumentation. The user only needs to implement the new cache simulation functions in the shared library used by the CacheSimulator.

In the current version of CacheSimulator, the simulator takes a list of memory hierarchies as input. Each memory hierarchy defines the number of levels in the memory hierarchy and properties of caches in each level. When collecting results, CacheSimulator does not differentiate between different memory instructions from the same block and writes results of cache hit/miss counts for each basic block and for each cache in the input list of memory systems.

Similar to the BasicBlockCounter tool, to relate the sequence numbers to unique hashcodes for the instrumented basic blocks, CacheSimulator also writes information about the blocks to an output file during instrumentation. Again, this information consists of the hashcode, sequence number, virtual address, function name, and source file name of the basic block.

3.4 Experimental Results

All experiments were performed on Datastar[10], a machine at the San Diego Supercomputer Center. The relevant part of this system is comprised of 171 P655+ SMP nodes, each node consisting of 8 processors that run at a clock rate of 1.5GHz and share 16GB of main memory. These nodes are connected with a Federation interconnect [11].

3.4.1 Instrumentation Time

We first present the total amount of time required to instrument real applications. The programs we used are HYCOM, GAMESS, AVUS, and WRF from the Department of Defense's Technology Insertion 2007[7] (TI-07) application workload. These applications exhibit varying static properties in areas such as number of functions, number of basic blocks and size and complexity of control flow graphs. For example, in WRF there are several functions for which the control flow graph for that function consists of more than 25K basic blocks.

Table 1 presents the amount of time spent to instrument applications using the BasicBlockCounter rewriting tool described earlier. It presents the function and basic block counts in each application. In addition to the total time spent to instrument executables, the table also shows the percentages of time spent during different stages of instrumentation.

Table 1 shows that the time spent to instrument applications ranges from 2 to 113 seconds. The table also shows that around 30% of the time is spent to generate program objects such as CFG and natural loops whereas around 50% of the time is spend for code injection for the applications other than WRF. For WRF, more time is spent generating program objects compared to the other applications (85%).

Even though the number of basic blocks in WRF is less than GAMESS, Table 1 shows that it takes significantly higher time to generate CFG and natural loops in WRF. This is mainly due to the fact that in WRF, there are several functions that have a large number of basic blocks (maximum around 50K) and complex CFGs. Since our loop generation algorithm [12] involves the use of dominator information, such complex and large control flow graphs require significantly higher amounts of time to generate this information. Overall, Table 1 shows that time required for instrumentation is proportional to some combination of the number of basic blocks in the applications as well as the complexity of the flow graphs.

3.4.2 Instrumentation Overhead

We also measured the slowdown caused by the BasicBlockCounter and CacheSimulator rewriting tools (we do not measure the slowdown of BasicBlockTracer because this depends largely on the efficiency of the I/O facilities being used) using applications HYCOM, GAMESS and AVUS from TI-07 application workload. We ran each application using the standard input sets for two CPU counts. We use a sampling rate of 10% for the CacheSimulator (i.e., we simulate only 10% of the addresses found in the address buffer) and we bound the maximum number of visits to each basic block to 50,000 visits, after which no samples are taken into consideration from that block. For the CacheSimulator experiments, we used the most frequently accessed basic blocks that correspond to the 95% of program execution for each application simulating. We simulated 19 different memory hierarchies containing two or three cache levels each.

Table 2 presents the results of our experiments where we measured the execution of the applications when running instrumented with BasicBlockCounter and CacheSimulator functionality. First and second columns in this table show the application and CPU count for the experiment. The third column presents the original execution times of the applications. The fourth and fifth columns present the execution time and slowdown ratio compared to the original execution times for the application running BasicBlockCounter instrumentation. Similarly, the sixth and seventh columns present the execution time and slowdown ratio for the application running CacheSimulator instrumentation.

Table 2 shows that BasicBlockCounter slows the applications down by a factor ranging from 1.22 to 1.48 (the average slowdown is 1.33), whereas CacheSimulator instrumentation simulating 19 cache structures slows down execution up to 7.29 times (the average slowdown is 5.78).

More importantly, Table 2 shows that both of the rewriting tools introduce acceptable instrumentation overhead. Moreover, the instrumentation overhead introduced by PMaCinst is comparable to the instrumentation overhead introduced by ATOM and is less than that of other dynamic instrumentation tools like PIN and Dyninst [8]. Intuitively, one would expect PMaCinst to introduce less overhead than PIN and Dyninst because both of these tools can disrupt the control flow of normal program execution in addition to any instrumentation code that is introduced. Like ATOM, PMaCinst performs all instrumentation prior to runtime. This means that it does not have some of the advantages that dynamic instrumentation tools have such as the ability to modify instrumentation at runtime, but it does have a significant performance advantage in the cases we have shown here.

Table 1. Time spent for instrumentation for BasicBlockCounter

	Function Count	Basic Block Count	Instru. Time (sec)	Percentage Time Spent			
				Parsing		Code Injection	Other
				XCoff Parsing	CFG Loop		
AVUS	489	31,587	**1.9**	15.6%	31.8%	46.9%	5.7%
HYCOM	243	28,105	**2.1**	14.9%	31.3%	50.0%	3.8%
GAMESS	4,435	383,715	**27.8**	18.8%	30.7%	47.8%	2.7%
WRF	3,383	315,499	**113.1**	3.4%	85.2%	10.7%	0.7%

Table 2. Slowdown ratios due to instrumentation for BasicBlockCounter and CacheSimulator

Application	CPU Count	Original Time (sec)	BasicBlockCounter		CacheSimulator	
			Execution Time	Slowdown	Execution Time	Slowdown
AVUS	64	2,793	3,625	**1.30**	20,153	**7.22**
	128	1,399	1,899	**1.36**	10.192	**7.29**
HYCOM	59	2,081	3,081	**1.48**	9,990	**4.80**
	124	1,050	1,448	**1.38**	6,289	**5.99**
GAMESS	64	4,859	5,925	**1.22**	23,026	**4.74**
	128	3,039	3,816	**1.26**	14,019	**4.61**
Mean				**1.33**		**5.78**

4. Future Work

Currently, for assembly code insertion (i.e., code that is not wrapped in a function) PMaCinst requires that the user have some knowledge of the PowerPC instruction set and assembly code since we do not provide the ability to generate assembly code from high level C-like statements on the fly. In addition to this, the C++ API that is provided to allow for users to create function call stubs and to extend the DATA section are written in such a way that the user must perform low-level operations such as counting the number of instructions in each call stub and putting function arguments into registers manually. These operations can be fairly easily automated; call stub generation can be automatic (as can counting the number of instructions used in a call stub), and argument passing can be done in a generic way that does not require user intervention or that the user have any knowledge of the function calling conventions of the underlying architecture.

Since code and call stub generation should be automatic, so should several simple optimizations to make sure that the instrumentation code remains efficient. For example, the code in a particular instrumentation function will often not kill every register. As a result, it is not necessary for the call stub surrounding the function to save and restore a register that is not killed by the function.

It has also been shown [8] that there are several techniques that can be used for optimization within an instrumentation tool framework, given that the resulting instrumentation tools require asynchronous access to program information. One example of such a technique is to include a mechanism within the instrumentation framework to buffer any application data that the user wishes to examine, then to examine the data only when the buffer is full or nearly full (as is done in the CacheSimulator tool). This technique reduces the cache pollution that often occurs when executing large instrumentation functions frequently during the application run. There is no reason that this mechanism cannot be built into an instrumentation tool, which can be used when the user determines that the instrumentation functions can be executed asynchronously with respect to data collected from the application.

Finally, the binary rewriting tools in the current release of PMaCinst are not safe for multi-threaded applications because of the method used to save and restore machine registers. Registers currently get put into the DATA section of the application, an area of memory that is shared by every thread in the application. With multiple threads using the same area of memory for register storage, one thread could overwrite the stored register values for another thread resulting in incorrect execution. To remedy this, each thread should save registers on its call stack, an area private to each thread.

5. Related Work

ATOM [1] was one of the first and has remained one of the more popular static binary instrumentation tools available. ATOM works in a way that is very similar to PMaCinst; instrumentation is performed on the compiled binary prior to runtime, meaning that any overhead due to code analysis and code generation is incurred outside of the instrumented application's runtime. Unfortunately, ATOM is available only for the Alpha platform. Since this processor is not being produced anymore, ATOM is no longer viable as a long-term solution for those who wish to perform static, efficient instrumentation on a RISC-based platform. Our hope is that PMaCinst will have similar functionality to ATOM, while maintaining ease of use and efficiency. We also hope to include several other useful features such as data flow analysis and tool-aided buffering for instrumentation.

Pin[3] is a dynamic binary instrumentation tool that uses a JIT-based (Just In Time compilation) approach to instrumentation. This approach entails running the application on top of Pin, while Pin intercepts the application at a natural control flow interruption in the program to perform instrumentation on the next part of the program. For efficiency, Pin does many things including caching these instrumented sequences of code to allow for re-use and chaining instrumented sequences of code together to avoid tool intervention.

Dyninst[2] is another popular dynamic instrumentation tool that uses a technique called code-patching to perform instrumentation. Similar in concept to what is done in PMaCinst, this technique replaces an instruction from the application with a jump instruction (which they call a trampoline) to the function call stub and instrumentation code. The key difference between PMaCinst and Dyninst is that Dyninst performs all code-patching at runtime instead of prior to runtime. This has several advantages, including the ability to insert, remove and customize instrumentation during runtime. But performing modification to the program at runtime also has a significant performance disadvantage, resulting in inefficient execution of the instrumented application.

6. Conclusions

In the work we introduced a binary instrumentation tool-building platform called PMaCinst, which allows for the insertion of arbitrary code into an XCOFF executable compiled for AIX on a PowerPC processor. The API provided by PMaCinst allows not only for the insertion of code, but also to extend the DATA section of an executable in order to store data that can be used to aid instrumentation routines.

We then described three example binary rewriting tools written using PMaCinst. BasicBlockCounter counts the number of times each basic block is executed during a program run. BasicBlockTracer prints detailed information about each basic block that is executed during a program run, and CacheSimulator simulates how the address stream of the application would perform on a number of real and theoretical cache structures.

Our experiments showed that the slowdown introduced by adding instrumentation through the PMaCinst API is comparable to the slowdown of similar instrumentation done by the ATOM toolkit. The current API still requires some knowledge of the underlying ISA, which is an issue we plan to address in the near future.

References

[1] A. Srivastava and A. Eustace. ATOM: A Flexible Interface for Building High Performance Program Analysis Tools. USENIX Winter Conference, January 1995.

[2] B. Buck and J. Hollingsworth. An API for Runtime Code Patching. Journal of High Performance Computing Applications 14 (4), Winter 2000.

[3] C. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. Reddi and K. Hazelwood. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Programming Language Design and Implementation (PLDI), June 2005.

[4] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia and A. Purkayastha. A Framework for Application Performance Modeling and Prediction. Supercomputing, November 2002.

[5] IBM Corporation, XCOFF Object File Format, http://publib.boulder.ibm.com/infocenter/pseries/v5r3/index.jsp?topic=/com.ibm.aix.files/doc/aixfiles/XCOFF.htm

[6] IBM Corporation, PowerPC User Instruction Set Architecture, http://www-128.ibm.com/developerworks/eserver/library/es-archguide-v2.html

[7] Department of Defense, High Performance Computing Modernization Program. Technology Insertion 07. http://www.hpcmo.hpc.mil/Htdocs/TI/.

[8] X. Gao, M. Laurenzano, B. Simon and A. Snavely. Reducing Overheads for Acquiring Dynamic Traces. International Symposium on Workload Characterization (ISWC), September 2005.

[9] D.H. Bailey and A. Snavely. Performance Modeling: Understanding the Present and Predicting the Future. EuroPar, September 2005.

[10] San Diego Supercomputer Center. Datastar User Guide. System Configuration. http://www.sdsc.edu/user_services/datastar/getstart.html#system.

[11] IBM Redbooks. An Introduction to the New IBM eServer pSeries High Performance Switch. http://www.redbooks.ibm.com/redbooks/pdfs/sg246978.pdf

[12] A.V. Aho, R. Sethi and J.D. Ullman. Compilers: Principles, Techniques and Tools. Addison-Wesley, 1986. ISBN 0-201-10088-6.

[13] T. Lengauer and R.E. Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. Transactions on Programming Languages and Systems (TOPLAS), July 1979.